Fluid Tool - Product Description

William Scherlis, Carnegie Mellon University, Feb 28, 2005

What is it? The Fluid Tool provides the working developer with a means to detect potential race conditions in Java programs and assure their absence. The Tool is integrated into the widely adopted Eclipse development environment. It is intended for use by individual developers actively evolving a Java code base, by software teams with build and configuration server, or by acceptance evaluators assessing code developed by others.

Features: The Fluid Tool is distinguished from other source-code analysis tools in four ways: (1) It provides positive assurance, in the sense that there are no false negatives—Fluid is able to assert an absence of errors with respect to a given model. (2) It addresses the hardest errors—the errors that defy conventional testing and inspection. The current version of the tool provides positive assurances for Java race conditions and a number of other critical dependability error types. These types of errors may be manifest in code at locations far from the actual enabling fault sites. Additionally, these errors may be intermittent and infeasible to replicate. (3) The tool enables developers to express critical design intent not expressible directly in code, for example, which locks protect which data. (4) The tool does not give false positives. This is due to the use of explicit design intent. Declarations of design intent are very simple—mostly one line javadoc-style annotations interspersed in program text. The tool uses a hybrid suite of composable static analyses to assess consistency of the code and the models expressed using these program annotations.

Benefits: The Fluid Tool provides static assurances for critical multi-threading properties that are difficult or impossible to assess using traditional testing, inspection, and runtime checking techniques. The analyses provide both “good news” and “bad news”—the tool can provide static assurance of absence of race conditions (with respect to models provided) as well as to detect potential races or other dangerous conditions in the code. As a developer evolves a system, the tool will track consistency of model and code in an ongoing fashion. It also detects potentially dangerous conditions such as lock-protected references to shared state that may not itself be protected. (This is usually repaired by extending the declaration of design intent to indicate that the lock is meant to protect the delegate object. The tool then automatically analyzes whether there are any potential "leaking" references to that object.)

Successes: The Tool has been experimentally applied to a wide range of at-scale Java production systems and components in aerospace, government, and top-10 software vendors. In virtually every case, including widely used library code, it has found faults that can trigger race errors (not false positives). The suggested corrections passed the internal vendor regression test suites. At multiple sites, suggestions were made by developers and first-line managers that the design intent notation be adopted on an ongoing basis for documenting concurrency-related design intent. Additionally, in several of these case studies, the Fluid team was able to rapidly locate faults that had eluded intense bug-finding effort. In one case, the tool located a fault that had been the target of an active search by the development team over a period of several weeks prior to the case study engagement. As developers made corrective changes to the code, the Tool supported a process of rapid iteration to experiment with adjustments to both code and model in order to achieve consistency.

Contexts in which it is best used: The Fluid Tool is robust and scalable, and has been applied to Java systems at a scale of several hundred KLOC. It works most effectively on systems that are

decomposed into subsystems—this allows for more rapid iteration of analysis. Code must build in the host tool environment (currently Eclipse) before it can be analyzed. Extensive or global use of meta-linguistic features such as reflection and highly specialized loaders may be problematic. The tool presently supports analyses for lock-based concurrency and non-lock concurrency, in which state is protected through policies that regulate which threads can touch which data (e.g., safe use of a GUI redraw thread). The non-lock concurrency analyses also apply to realtime threading policies, for example, statically assuring that no-heap threads respect policies. Additional error types are being added, and it is possible to develop specialized analyses, though that requires a more strategic engagement. The experience of the several case study engagements already accomplished indicates that an early "diagnostic" collaboration of one to three days that brings Fluid team members together with developer/users is the most effective way to initiate use.

Compare with alternate known products or technologies. There are several categories of technologies with which Fluid can be compared:

(1) Testing tools. For many categories of faults and errors, test tools can be extremely effective in reducing defects. But when systems are non-deterministic and asynchronous, errors can be intermittent and infrequent, and even then manifest only due to particular combinations of simultaneous stimuli. In these cases, testing is unlikely to be a successful approach to identifying the faults that lead to intermittent errors such as races and deadlocks. Additionally, testing cannot generally be used to identify inappropriate access to critical state in a system, because the kind of alias analysis required yields a result of universal character.

(2) Inspection, including tool-assisted inspection. Inspection is effective in assessing design and local code manifestations, such as pattern compliance. The errors addressed by Fluid, however, tend to have a distributed and often global character. Error manifestations may be far from the enabling fault sites. Additionally, code inspection is unlikely to juxtapose multiple code segments to reveal an inconsistency in intent (e.g., which lock should be used) when the intent is not expressed directly by developers.

(3) Model checking tools. Model checking is an exceptionally powerful technology that can detect many kinds of concurrency errors, but in the current state of the art scalability is a major challenge, as is the modeling of design intent. Scalability is elusive because it is still a research problem to compose assurance results, which is a key to the scalability of Fluid. Model checking can yield a positive assurance (i.e., a guarantee of no false negatives) only for certain categories of models and systems.

(4) Rule-based bug finding tools. Many rule-based bug finding tools match a library of patterns against the syntactic structure of code, as represented by abstract syntax trees. Some augment this by static analyses of various kinds, ranging from binding and typing to more sophisticated analyses. But in general these tools yield false positives because design intent is guessed and the analyses are imprecise. Fluid increases precision due to a combination of design intent, the use of composition and cut-points in analysis, and an aggressive specialization and hybridization of the analyses used to achieve the overall aggregate assurances. Rule-based tools generally also yield false negatives, for the same reason.

(5) Specialized analysis tools. There are several specialized analysis tools in the market, particularly for C and C++, that address language-specific issues related to storage safety, type safety, and respect for encapsulation. There are some emerging analysis tools for Java.

(6) Program verification systems. Program verification has been a research topic for nearly 40 years (dating back to Floyd's landmark paper), and is useful for very small and highly critical code components. Unfortunately, scalability has continued to elude verification approaches based on traditional program assertions and general-purpose theorem proving. Some "compromise" approaches have been developed (e.g., ESC/Java) that focus on particular characteristics such as array bounds and other code-safety issues. These approaches can operate at greater scale and on existing code bases, but they require extensive program annotation with pre- and post-conditions and invariants before analysis results are forthcoming. Fluid focuses on error types that do not require this kind of extensive *a priori* modeling.

(7) Dynamic (runtime) monitoring. Runtime monitoring is a useful technique to detect certain categories of errors at runtime. The technique has the advantage of detecting errors, and so allowing for remediation and recovery. Indeed, this is part of the idea of "self-healing architectures," in which pervasive logging and auditing supports rapid detection of anomalies. Certain kinds of errors cannot easily be detected at runtime, however, without radical redesign of a system. For example, inappropriate aliasing can often be detected only at great expense (reference counting). Certain kinds of races may not be detectable at all.

Finally, we note that the present version of Fluid supports Java only, while there are many tools in the market that support C and C++.

What will a successful collaboration look like?

What will the technology provider do? We propose a pattern of engagement that has an initial phase of collaboration between the engineering organization and the Fluid team, followed by more independent use by the engineering organization with ongoing technical support from the Fluid team. The Fluid team will collaborate with engineering organizations developing or maintaining Java code. Generally speaking, a direct or closely coordinated collaboration for an initial period of two to four days is the most effective start. During this collaboration, some initial modeling is accomplished, races may be found, and assurances provided. This provides *in situ* tool training to a number of members of the engineering organization. The Fluid team then provides a moderate level of usage support and periodic enhancements.

What should the development team do? The NASA development team should communicate with the Fluid team to assess suitability and needs. The NASA development team should appoint one or two members to serve as principal technical liaisons with the Fluid team. It is desirable, but not necessary, for the NASA development team to partition large Java system into a set of subsystems of about 50-100KLOC each. This will enable a more rapid iteration cycle on code development, modeling, and analysis. An ideal development organization will already be familiar with Eclipse (www.eclipse.org). If not, the subsystems to be analyzed need to be able to be built in Eclipse. (We have not experienced any major problems in this regard—Eclipse supports many build tools and patterns, including ant and gmake.) It is expected that Fluid will be deployed on other development platforms, depending on demand. Finally, the development team should provide feedback to the Fluid team regarding usability, coverage, features, etc.

How will the technology provider work together with the development team to ensure a successful collaboration? The Fluid team will support an initial intensive period (typically a couple of days) of collaboration. This will assure that the tool can be integrated into the development environment, that the Java subsystems of interest can build, and that models can be defined for critical attributes.